



# **CPEN 311: Digital Systems Design**

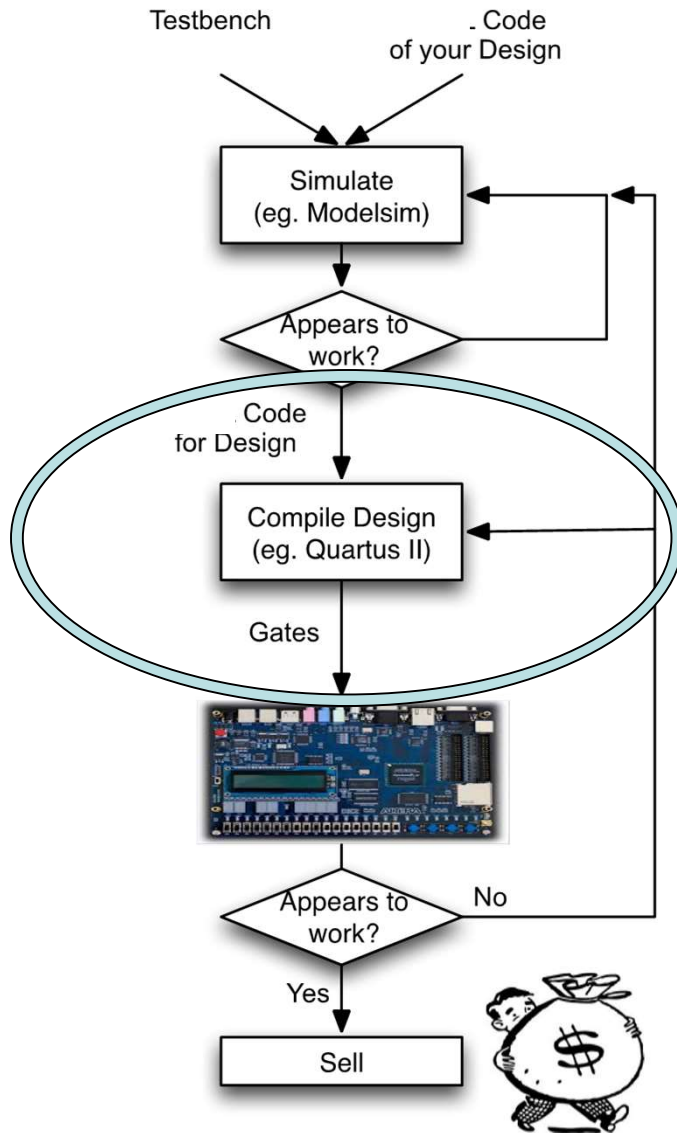
## **Synthesizable Verilog**

# CPEN311

---

- Delivery
  - Lectures – Cristian Grecu ([grecuc@ece.ubc.ca](mailto:grecuc@ece.ubc.ca))
  - Canvas – everything, including links to:
    - Zoom – lectures (recorded)
    - Piazza – discussion
    - GitHub Classroom – labs (DE1-SoC + simulation)
- Marks
  - Quizzes 5%, multiple choice in Canvas (async with deadlines!)
  - Labs 55%, submitted to GitHub for autograding + “interview/demo”
    - Lab 0 ungraded May 21 (set up tools, submit github userID)
    - Lab 1 15% May 21
    - Lab 2 20 June 4
    - Lab 3 20% June 17 (last day of classes, NOT Friday)
  - Midterms 15% (during Thurs lecture, date TBA)
  - Exam 25%, must pass exam to pass course

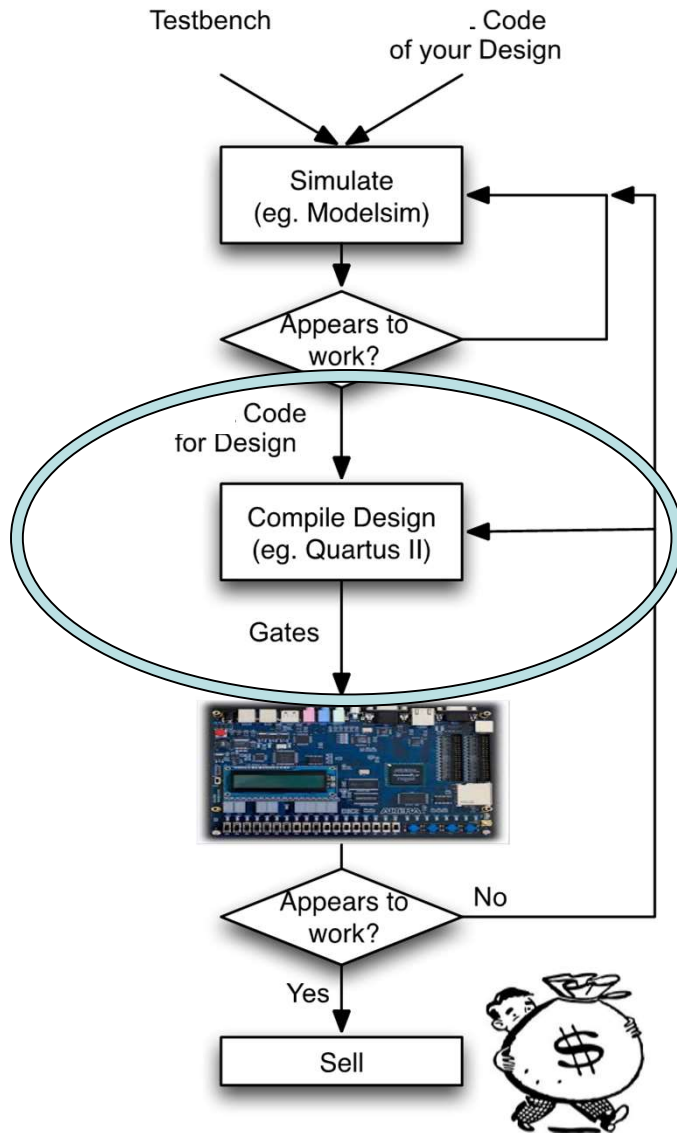
# Verilog Flow



The synthesis tool (eg. Quartus II) “compiles” the **Verilog code** into **gates**, making hardware out of the code.

The gates are implemented directly on an FPGA chip by placing logic functions inside the chip and wiring them together.

# Verilog Flow



**What do you think happens if the synthesis tool can not make hardware for your Verilog?**

**The gates will not implement the behaviour that you specified...**

**Your circuit will not implement the behaviour that you specified...**

# Introduction to this Slide Set

---

**In Lab 1, you are writing a fairly small Verilog Description**

For larger designs, if you aren't *\*really\** careful, it is likely that when you try to compile your code to hardware, **it won't work!**

Why?

I'll tell you in this slide set, and also talk about how to make sure it does work.

**Why is this so important?**

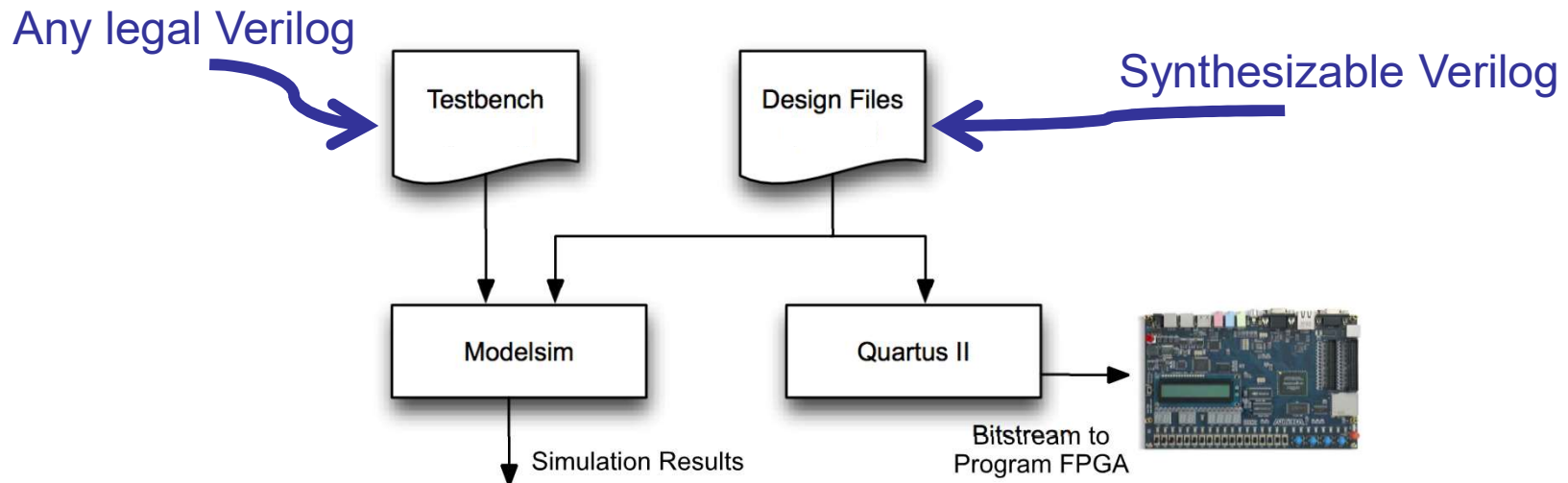
Because, it will save you tons of debugging time in the lab if you understand it.



# Recall: What is Verilog?

## Verilog serves two roles:

- Synthesis – Describe hardware that you ultimately want to create
- Simulation – Describe hardware for simulation, and describe tests



Subset of Verilog that can be synthesized is called **synthesizable**

- Many legal Verilog constructs **cannot be synthesized**

# Synthesizable Language Constructs

---

- Entities and Architectures / Modules
- Signals / Wires and Regs
- Concurrent Signal Assignments
- Module Instantiations
- Processes / Always blocks
  - If/else Conditional Statement
  - Case Statement

These are generally synthesizable ...

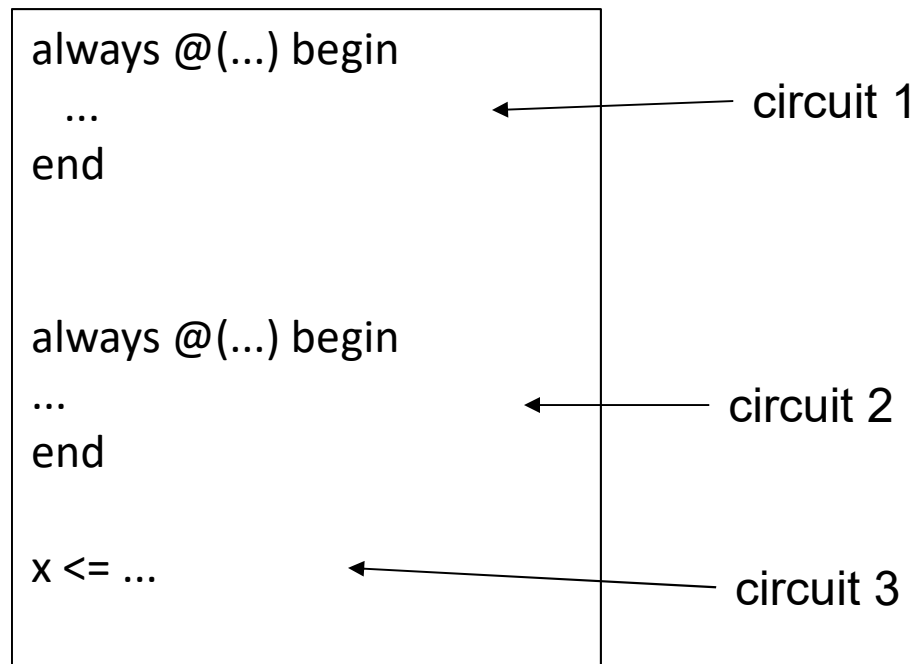
**To be Synthesizable, there needs to be a way to determine an equivalent gate-level implementation of the construct**

For all of the constructs above, there is a straight-forward “recipe” to determine the gate-level implementation ...

**...except for the ALWAYS construct**

A modern synthesis tool:

- Extracts processes and concurrent assignments from the code
- Converts **each** *process/always and concurrent assignment* to **a piece of hardware**



**Each process  
or concurrent  
assignment  
is treated  
separately !!**

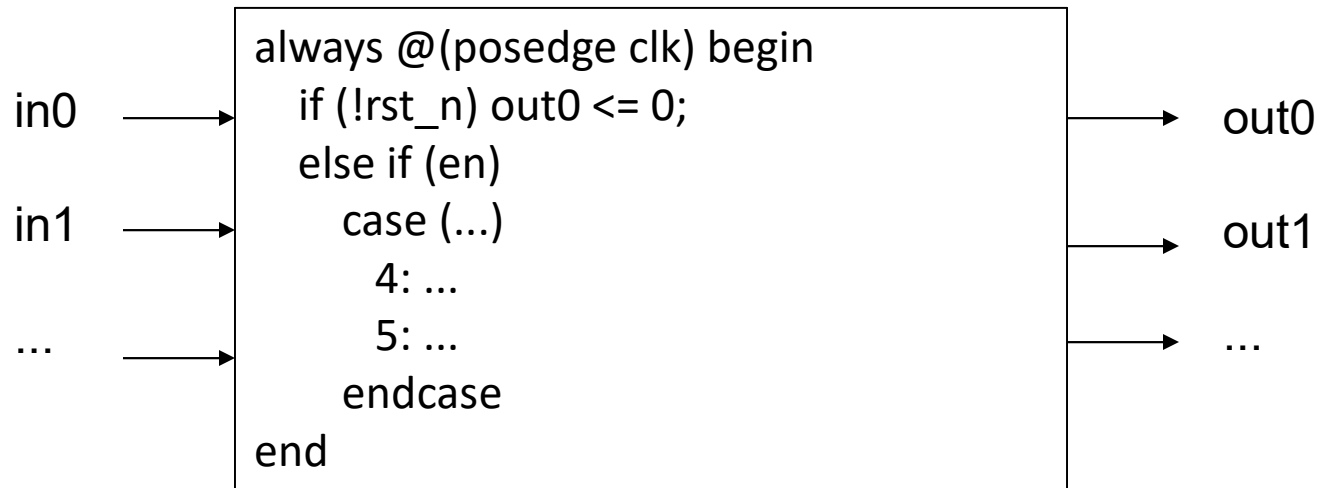


# Synthesis of Always Blocks

---

We use the Verilog **always** block to **model** the behaviour of a block of hardware.

The synthesis tool then **tries its best** to find a hardware implementation that matches this behaviour.



**It's possible to write a process that is not synthesizable**

# Three Coding Patterns

---

Synthesis tools use **PATTERN MATCHING** on each process to determine its hardware implementation

Three patterns that **ALL** synthesis tools can understand

1. **Purely Combinational**
2. **Sequential**
3. **Sequential with asynchronous reset**

**ANY PROCESS THAT YOU WRITE  
MUST USE  
ONE OF THESE THREE PATTERNS**

# **Pattern 1. Purely Combinational**

# 1. Purely Combinational

---

Process outputs are a function of the current input values

## Rule 1A:

- Every input to the process must be in the sensitivity list (or in Verilog 2001 you can use \*, or in SystemVerilog **always\_comb**)

## Rule 1B:

- Every output must be assigned a value for every possible combination of inputs
- In other words, every output must be assigned a value for every possible path through the process's description

```
always @(A or B or SEL)
    if (SEL == 1)
        C = A;
    else
        C = B;
```

# 1A. Sensitivity List Rule

---

## Rule 1A:

- Every input to the process must be in the sensitivity list

If you break this rule, you are saying...

... “If an event occurs on the input signal, do not immediately update the output”

From Quartus:

```
always @(A or B)
  if (SEL == 1)
    C = A;
  else
    C = B;
```

Warning (10235): Verilog HDL Always Construct warning at temp.v(5): variable "SEL" is read inside the Always Construct but isn't in the Always Construct's Event Control

Quartus is actually smart enough in this case, but don't rely on this in general. And your simulations will definitely be wrong.

In SystemVerilog, you can use **always\_comb** to ensure this

```
always_comb
  if (SEL == 1)
    C = A;
  else
    C = B;
```

# 1B. Output Assignment

---

## Rule 1B:

- Every output must be assigned a value for every possible combination of input values
- In other words, every output must be assigned a value for every possible path through the process' description

## If you break this rule, you are saying...

... “For the combination of input values where we don't assign a value to the output, the output should remember its old value.”


→ **Memory is implied → Sequential, not combinational**

## 1B. Output Assignment

---

Verilog:


```
always @(A or B or SEL)
  if (SEL == 1)
    C = A;
```



Warning (10240): Verilog HDL Always Construct warning at temp.v(4): inferring latch(es) for variable "C", which holds its previous value in one or more paths through the always construct

System Verilog:

```
always_comb
  if (SEL == 1)
    C = A;
```



Error (10166): SystemVerilog RTL Coding error at temp.v(5): always\_comb construct does not infer purely combinational logic.

Gives an error instead of a warning! Good reason to use always\_comb



## 1B. Output Assignment

---

### Rule 1B:

- Every output must be assigned a value for every possible combination of input values.
- In other words, every output must be assigned a value for every possible path through the process' description

The following works though:

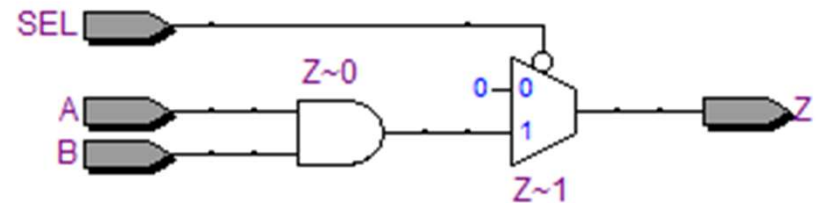
```
always @(A or B or SEL)
begin
    C = B
    if (SEL == 1)
        C = A;
end
```

**Make sure you understand why!**

# Inferred Latches

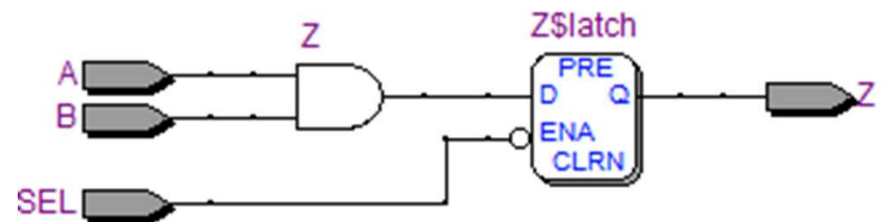
Say you were trying to describe the following combinational circuit

```
always @(A or B or SEL)
begin
    if (SEL == 1)
        Z <= A and B;
    else
        Z <= 0;
end
```



If we ignore this rule, look at the hardware that is generated:

```
always @(A or B or SEL)
begin
    // Note: missing ELSE
    if (SEL == 1)
        Z <= A and B;
end
```



A **level-sensitive LATCH** is inferred instead!

**This is not the behaviour we wanted!**

# CASE Statement

---

Careful to cover all choices in a CASE statement:

```
always @(A or B or C or SEL)
  case (SEL)
    2'b00: F = A;
    2'b01: F = B;
    2'b10: F = C;
  endcase
```

Warning (10270): Verilog HDL Case Statement warning at temp.v(6):  
incomplete case statement has  
no default case item

**BEST PRACTICE:** Use **default** to catch all other cases 19

# Summary 1. Purely Combinational

---

Process outputs are a function of the current input values

## Rule 1A:

- Every input to the process must be in the sensitivity list (or **always\_comb** in SystemVerilog, or use \* in Verilog 2001)

## Rule 1B:

- Every output must be assigned a value for every possible combination of inputs
- In other words, every output must be assigned a value for every possible path through the process's description

```
always @(A or B or SEL)
    if (SEL == 1)
        C = A;
    else
        C = B;
```

## **Pattern 2. Sequential**

## 2. Sequential

---

Each output changes ONLY on the rising or falling edge of a single clock

### Rule 2A:

- Only the clock should be in the sensitivity list

### Rule 2B:

- Only signals that change on the **same edge** of the **same clock** should be part of the **same always block**

```
always @(posedge CLK)  
    <logic>
```

**Sequential Circuit with Synchronous Reset  
Falls Under This Category**

...of the .../Documents/teaching/spens .../run2 .../wrong/temp/temp

File Project Assignments Processing Tools Window Help

Search altera.com

temp

Compilation Report - temp temp.v\*

```
1 module temp (D, CLK, Q);
2 input D, CLK;
3 output reg Q;
4
5 always @(D, posedge CLK)
6     Q <= D;
7 endmodule
```

IP Catalog

- Installed IP
  - Project Directory
    - No Selection Available
  - Library
    - Basic Functions
    - Bitec
    - DSP
    - Interface Protocols
    - Memory Interfaces and Controllers
    - Processors and Peripherals
    - University Program
- Search for Partner IP

Files

Task

- Compile Design
- Analysis & Synthesis
- Fitter (Place & Route)

Include D in sensitivity list for a flip-flop

Message

```
122 Verilog HDL Event Control error at temp.v(5): mixed single- and double-edge expressions are not
153 Can't elaborate top-level user hierarchy
    Quartus II 64-Bit Analysis & Synthesis was unsuccessful. 2 errors, 0 warnings
001 Quartus II Full Compilation was unsuccessful. 4 errors, 0 warnings
```

## 2. Sequential

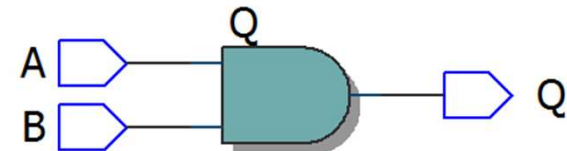
---

In SystemVerilog, you can use **always\_ff**

```
always_ff @(posedge CLK)
    Q <= D;
```

**be Careful:** Quartus II will not make a FF just because you use `always_ff`:

```
always_ff @(A or B)
    Q = A & B;
```



In this case, the user probably wanted a flip-flop, but got a combinational AND gate *with no warning* in Quartus II



**Aside: RESET SIGNALS:**

**Why do we need resets?**

# Flip Flop Reset Signals

---

A flip-flop can have either a synchronous or asynchronous reset

**Asynchronous Reset:** When the reset signal is high, the flip-flop is reset (forced to '0' ) immediately, regardless of the clock.

**Synchronous Reset:** On a rising clock edge, if the reset signal is high, The flip-flop is reset (forced to '0' ).

## The difference

Synchronous reset → flip-flop only resets on a rising clock edge

Asynchronous reset → flip-flop resets immediately

Recall rule for Pattern 2: Each output changes ONLY on the rising or falling edge of a single clock

Do either of these match this pattern?

**Asynchronous Reset:** When the reset signal is high, the flip-flop is reset (forced to '0' ) immediately, regardless of the clock.

**Synchronous Reset:** On a rising clock edge, if the reset signal is high, The flip-flop is reset (forced to '0' ).

# Describing DFF with Synchronous Reset

**Synchronous Reset:** On a rising clock edge, if the reset signal is high, The flip-flop is reset (forced to '0').

**Reset  
Case**

{

```
always @(posedge CLK)
  if (RESET == 1)
    Q <= 0;
  else
    Q <= D;
```

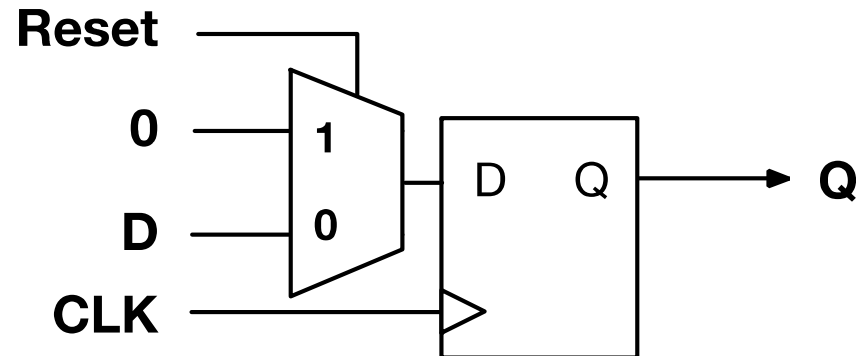
}

**Normal Operation**

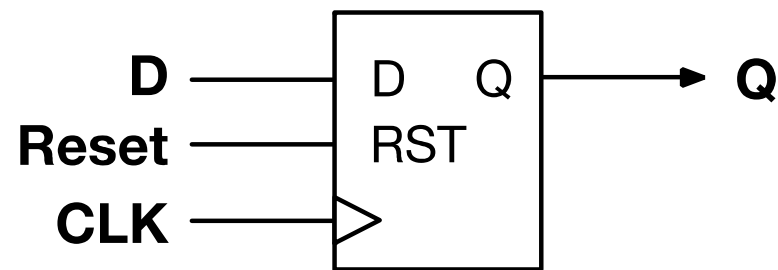
Reset behaviour described **inside** rising clock edge case<sub>28</sub>

# DFF With Synchronous Reset

Behaviour is equivalent to:



But there are better transistor level implementations of such functionality and we just assume that it's possible to create such a flip flop with synchronous reset input



**Asynchronous Reset:** When the reset signal is high, the flip-flop is reset (forced to '0' ) immediately, regardless of the clock.

?

**Synchronous Reset:** On a rising clock edge, if the reset signal is high, The flip-flop is reset (forced to '0' ).

✓ **Pattern 2 (Sequential)**

## **Pattern 3. Sequential with Asynchronous Reset**

### 3. Sequential with Asynchronous Reset

---

Reset occurs immediately

#### Rule 3A:

- Sensitivity list includes clock and reset

```
always @(posedge CLK or posedge RESET)
    if (RESET == 1)
        <reset assignments>
    else
        <normal operation/logic>
```



# Describing DFF with Asynchronous Reset

**Asynchronous Reset:** When the reset signal is high, the flip-flop is reset (forced to '0') immediately, regardless of the clock.

Verilog:

**Reset Case**

```
always @(posedge CLK or posedge RESET)
{
    if (RESET == 1)
        Q <= 0;
    else
        Q <= D;
}
```

**Normal Operation**

**RESET now in  
sensitivity list**

**Final Rule** (the most important rule of all):

**If you want to synthesize your circuit, every process must fall *exactly* into one of these categories. Every process. Every single one. No exceptions.**

**If one of your processes doesn't, you need to break it up into blocks, where each block does fit into one of these categories.**

(some synthesizers handle some other patterns, but you can't rely on that)

**IF YOU DON'T FOLLOW THESE  
THREE PATTERNS**



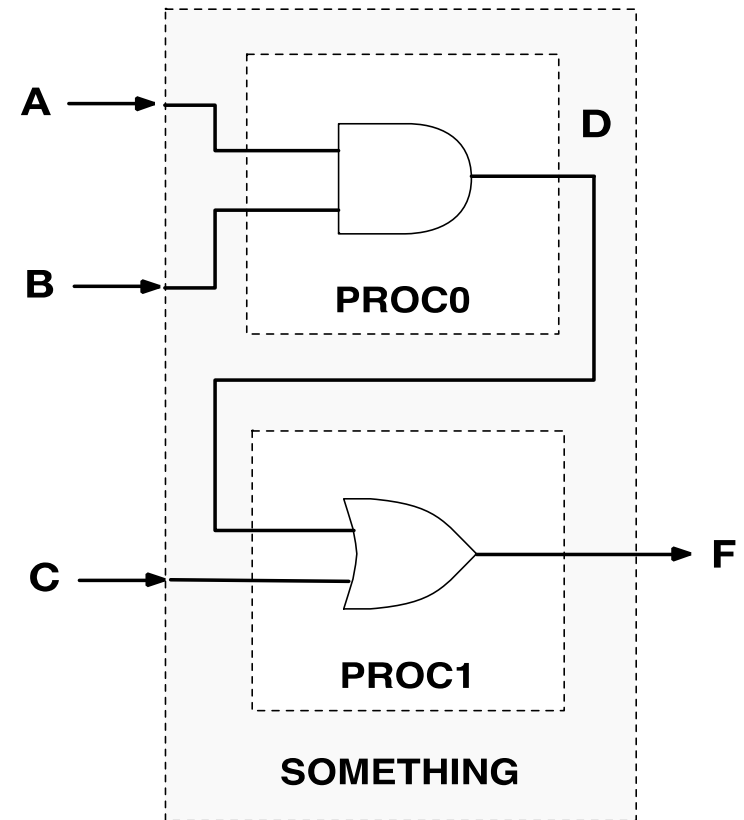
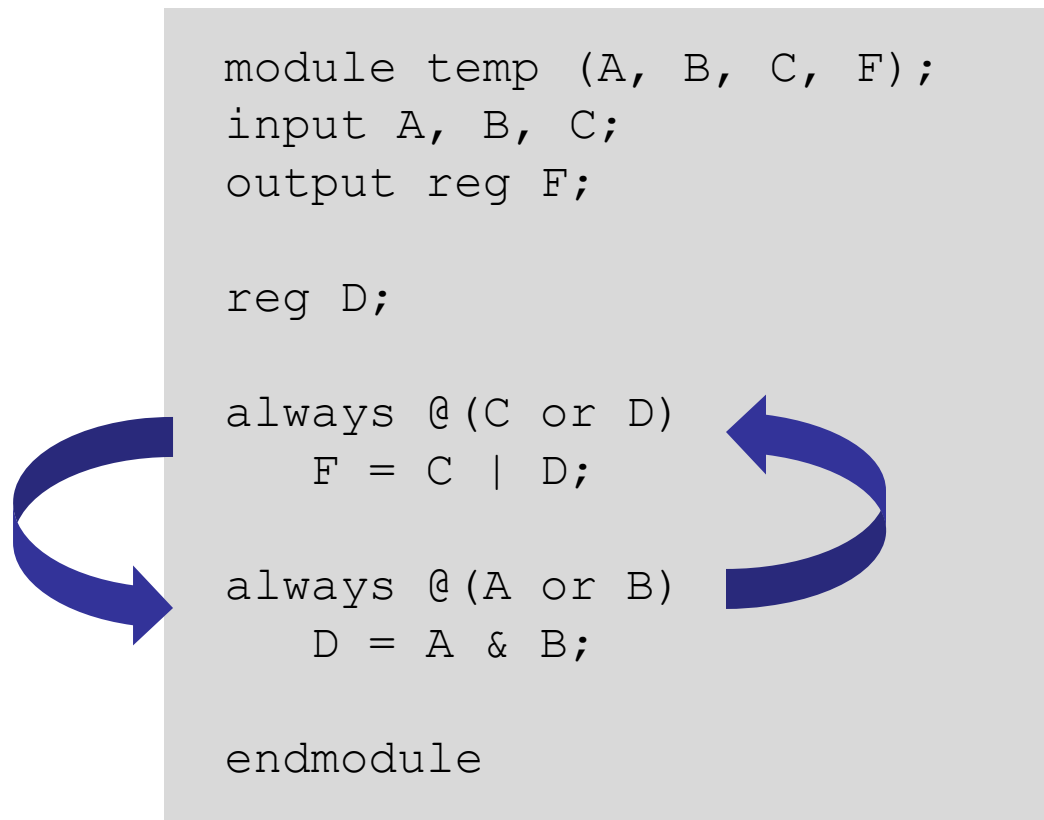
**YOU'RE GONNA HAVE A BAD TIME**

**Two more things to remember  
about always blocks**

# 1. Processes are Concurrent Statements

The order of CONCURRENT STATEMENTS don't matter.

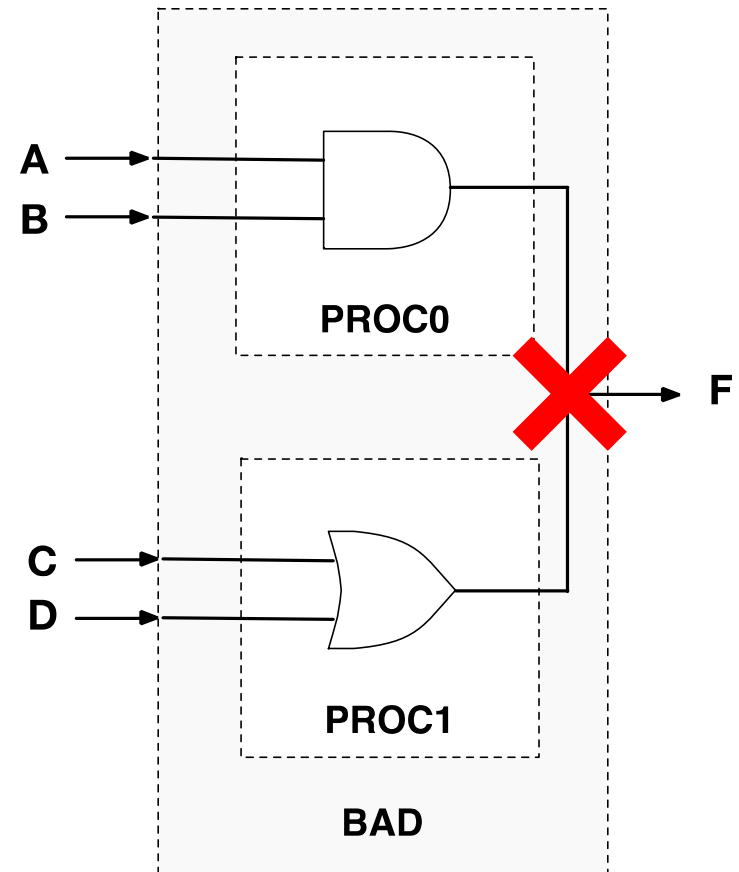

**Changing order of always blocks won't change synthesis result**



## 2. Don't Drive A Signal From Multiple Processes

Remember: can't drive the same wire with multiple gates.

```
module SOBAD (A, B, C, D, F);  
  input A, B, C, D;  
  output reg F;  
  
  always @(C or D)  
    F = C | D;  
  
  always @(A or B)  
    F = A & B;  
  
endmodule
```



# Summary of this Slide Set

---

All synthesizable always blocks should fall into one of three categories:

1. Combinational
2. Sequential
3. Sequential with Asynchronous Reset

If your always block doesn't follow one of these patterns, the tool will be unlikely to be able to synthesize it -> **Hardware likely won't work!**

In that case, you have to break it up

- We will talk about this in subsequent slide sets

We also talked about:

- Synchronous vs. Asynchronous Reset
- Two more things about Always Blocks